Санкт-Петербургский государственный университет

Кафедра технологии программирования

Подусенко Альберт Олегович

Выпускная квалификационная работа бакалавра

Разработка мобильного приложения, позволяющее делать фотографии при помощи распознавания мимики лица

Направление 010400

Прикладная математика и информатика

Научный руководитель,

Старший преподаватель

Малинин К. А.

Санкт-Петербург

2016

Содержание

Введение

Прошло уже больше сорока лет с тех пор, как компания Xerox в 1973 году представила первый персональный компьютер Xerox Alto с графическим пользовательским интерфейсом, поддерживающий устройство, которое определило то, как человек будет взаимодействовать с компьютером на протяжении долгого времени – компьютерную мышь. С тех пор человеко-машинные интерфейсы потерпели множество изменений, но тем не менее создание естественных, простых в управлении интерфейсов для различных устройств и приложений по-прежнему остается актуальной научной задачей. Благодаря стараниям ведущих ИТ-компаний развиваются различные решения, позволяющие человеку бесконтактно взаимодействовать с различными устройствами. Например, компания Microsoft разработала игровой контроллер Kinect, позволяющий пользователю управлять игровой консолью Xbox без помощи контактного игрового контроллера через устные команды, позы тела и жесты. Интересным направлением является и создание естественных пользовательских интерфейсов для смартфонов. Несмотря на то, что уже на протяжении нескольких лет сенсорный дисплей остается основным средством взаимодействия человека со смартфоном, компании Apple и Google активно развивают своих персональных программных агентов, которые предоставляют пользователю возможность “общаться” со смартфоном при помощи голосовых команд. Равным образом южнокорейская компания Samsung внедрила в свои смартфоны функцию Air Gesture, которая предоставляет пользователю возможность управления устройством при помощи жестов.

Интерес, который проявляют ИТ-компании к созданию новых способов взаимодействия со смартфоном вполне оправдан. Ведь ни для кого не секрет, что смартфоны стали неотъемлемой частью нашей жизни. Они помогают нам экономить наше время, обеспечивая доступ практически к любой информации в нужный момент. С помощью смартфонов люди обмениваются фотографиями, выкладывая их в различные социальные сети или передавая их

друг другу через современные мессенджеры. Согласно данным techinfographics[1] пользователи социальных сетей загружают более одного миллиона селфи[1] каждый день. Наличие фронтальной камеры чуть ли не у каждого современного телефона еще больше популяризовала процесс съёмки селфи. Однако тенденция к увеличению габаритов современных мобильных устройств приводит к следующему неудобству: из-за больших размеров телефона, пользователю становится тяжело дотягиваться до кнопки спуска затвора фотокамеры, не уронив при этом сам телефон. А в холодное время года процесс съемки селфи усложняется еще и тем, что большинство современных смартфонов оснащены ёмкостными сенсорными экранами, которые не реагируют на прикосновения в перчатках, поэтому для создания селфи владелец смартфона вынужден снимать перчатки. Таким образом, привычный интерфейс взаимодействия человека и фотокамеры мобильного устройства становится недружелюбным.

Желая найти решение, которое бы упростило процесс съемки для всех любителей селфи, было обращено внимание на мимику человека, а вернее на её распознавание. Ввиду своей простоты и выразительности в качестве основы для мобильного приложения, позволяющего делать фотографии, были выбраны такие мимические движения, как подмигивание и улыбка

[1] Селфи (англ. selfie, от “self” – сам, себя) – разновидность автопортрета, заключающаяся в запечатлении самого себя на фотокамеру.

Постановка задачи

Целью данной выпускной квалификационной работы является разработка мобильного приложения, которое позволит создавать фотографии при помощи распознавания мимических движений. Для этого в рамках выпускной квалификационной работы должны быть решены следующие задачи:

1. Изучить теорию для решения задач распознавания мимических движений
2. Выбрать мобильную операционную систему для написания приложения
3. Разработать алгоритм позволяющий фиксировать состояние глаз и детектировать улыбку на лице
4. Протестировать реализованный алгоритм
5. Реализовать разработанный алгоритм в мобильном приложении

Глава I. Теоретическая часть

Данная глава посвящена изучению методов, которые лягут в основу приложения, позволяющего создавать фотографии при помощи распознавания мимических движений. В качестве основного метода распознавания мимики рассматривается метод Виолы-Джонса. Также в рамках данной главы описаны такие методы, как детектор границ Кэнни и преобразование Хафа, которые будут использованы для детектирования подмигивания. Помимо перечисленных методов будут рассмотрены современные мобильные операционные системы и средства разработки мобильных приложений.

1.1 Детектор границ Кэнни

В 1986 году профессор Массачусетского технологического института Джон Кэнни (John F. Canny; 1953 г.) разработал оператор обнаружение границ изображения – детектор границ Кэнни или алгоритм Кэнни[2]. Данный алгоритм состоит из пяти отдельных шагов:

1. Сглаживание
2. Поиск градиентов
3. Подавление немаксимумов
4. Двойная пороговая фильтрация
5. Трассировка области неоднозначности

Перейдем к подробному описанию каждого шага.

1.1.1 Сглаживание

Все снимки получаемые при помощи камеры неизбежно содержат некоторое количество шума, который устраняется посредством размытия изображения. В рассматриваемом алгоритме для размытия используется фильтр Гаусса с ядром $N \times N$. Данный фильтр представляет собой матрицу свертки, заполненную по закону нормального распределения.

1.1.2 Поиск градиентов

На данном шаге происходит поиск направлений смены яркости на изображении при помощи оператора Собеля. Сначала исходное изображение A сворачивается по оси X и по оси Y при помощи следующих фильтров:

$$K_{GX} = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, \qquad K_{GY} = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}.$$

То есть

$$G_X = K_{GX} * A, \qquad G_Y = K_{GY} * A,$$

где * – операция свертки.

После чего вычисляется приближенное значение градиента и его направление:

$$G = \sqrt{G_X^2 + G_Y^2}, \qquad \theta = arctg\left(\frac{G_Y}{G_X}\right).$$

Потенциальными границами на изображении A выбираются точки, в которых градиент имеет наибольшее значение.

1.1.3 Подавление немаксимумов

При подавлении немаксимумов происходит поиск пикселей, в которых достигается локальный максимум градиента в направлении вектора градиента. Рассмотрим пример, изображенный на рисунке 1:

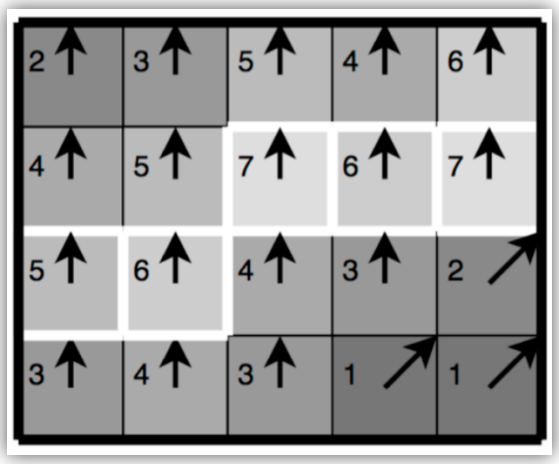


Рис. 1. Выбор локального максимума

Большая часть пикселей из примера имеют направление на север, поэтому значения градиента в этих точках сравниваются со значениями градиента нижерасположенных и вышерасположенных пикселей. Таким образом, обведенные белым контуром пиксели останутся в результирующем изображении, остальные будут подавлены.

1.1.4 Двойная пороговая фильтрация

Большая часть пикселей, оставшихся после предыдущего шага определяют границы изображения, тем не менее в алгоритме Кэнни применяется пороговая фильтрация, которая позволяет исключить пиксели, которые вызваны шумом исходного изображения. Детектор границ Кэнни использует два порога фильтрации, значения которых лежат в промежутке от 0 до 255. Если значение пикселя выше верхней границы – он принимает максимальное значение (граница считается достоверной), если ниже – пиксель подавляется. Пиксели, значения которых попадают в диапазон между порогами, принимают фиксированное среднее значение.

1.1.5 Трассировка области неоднозначности

На последнем шаге анализируются группы пикселей, попавших в среднюю область. Пиксели, лежащие по соседству с границей, присоединяются к ней, остальные – отбрасываются.

1.2 Преобразование Хафа

Преобразование Хафа[3] – это алгоритм поиска линий, кругов и других простых форм на изображении. Идея алгоритма состоит в представлении искомого объекта в виде параметрического уравнения, параметры которого образуют фазовое(аккумуляторное) пространство. После, берется бинарное изображение(результат работы детектора границ Кэнни), перебираются все пиксели границ и делается предположение о том, что каждый взятый пиксель принадлежит искомой кривой. Так, для каждого пикселя изображения рассчитывается нужное уравнение и получаются необходимые параметры,

которые сохраняются в фазовом пространстве Хафа. Далее происходит обход пространства Хафа и выбор максимальных значений, за которые проголосовало больше всего пикселей картинки, что и дает параметры для уравнений искомого объекта. В данной работе преобразование Хафа понадобится для детектирования радужной оболочки глаза, которая имеет форму окружности. Поэтому рассмотрим работу данного алгоритма для поиска окружностей.

Как известно точки окружности можно представить следующей формулой:

$$(x-a)^2+(y-b)^2=R^2,$$

где (a,b) – координаты центра окружности, а R – ее радиус. Уравнение семейства окружностей имеет вид:

$$F(a,b,R,x,y)=(x-a)^2+(y-b)^2-R^2 \qquad (1.1)$$

Как видно из уравнения (1.1), для поиска окружностей нужно задавать три параметра: две координаты центра и радиус. Так как размерность фазового пространства сказывается на эффективности использования преобразования Хафа, желательно каким-либо образом минимизировать количество параметров в (1.1). Например, можно определить возможные значения радиусов искомой окружности $R_1..R_2$, тем самым уменьшив пространство Хафа на единицу. Так, каждой точке (x,y) изображения ставится в соответствие набор точек фазового пространства (a,b). Для каждой точки (a_0,b_0) фазового пространства (a,b) создается счетчик, отвечающий за количество точек (x,y), лежащих на окружности с параметрами (a,b). Непрерывное фазовое пространство переводится в дискретное, путем наложения сетки на пространство (a,b), одной ячейке которого соответствует набор кривых с близкими значениями a и b. Параметры a и b ячейки характеризуются как средние значения точек a_i и b_i, попавших в эту ячейку. Для каждой белой точки изображения ищутся ячейки в фазовом пространстве (a,b) и увеличивается счетчик. После обхода всех точек бинарного изображения, фазовое пространство заполнится. Далее будут выбраны

окружности, которые содержат наибольшее количество точек изображения. Такой результат достигается путем ввода порога: ячейки, значение счетчиков которых выше этого порога объявляются истинными окружностями — остальные отбрасываются). Однако, такой подход чреват появлением окружностей, которые имеют “почти одинаковый” центр. Такие окружности следует исключить следующим образом:

1. Найти ячейку с максимальным значением счетчика. Если это значение меньше порога, выход;
2. Окружность соответствующую найденной ячейки (a', b') принять за истинную;
3. Найти белые точки на исходном бинарном изображении, лежащие на найденной окружности.
4. Найденные точки перекрасить в черный цвет, и для каждой окружности проходящей через эти точки найти соответствующую точку в фазовом пространстве (a, b) и уменьшить счетчик соответствующей ячейки на единицу.
5. Перейти к шагу 1.

1.3 Метод Виолы-Джонса

Метод Виолы–Джонса[4] – алгоритм, позволяющий детектировать объекты в реальном времени. Этот метод был разработан в 2001 году усилиями Пола Виолы и Майкла Джонса и сразу же стал прорывом в области обнаружения объектов. Несмотря на то, что основной целью при разработке метода Виолы-Джонса было обнаружение лиц, алгоритм равно хорошо работает и распознает различные классы изображений, в том числе и черты лица при различных условиях освещенности под углом до 30 градусов[5]. Являясь одним из лучших по соотношению показателей эффективность распознавания и скорость работы, метод Виолы-Джонса обладает низкой вероятностью ложного срабатывания. Данный метод использует принцип скользящего окна, суть которого заключается в выборе некоторого

прямоугольного окошка на изображении, например, 24x24 пикселя. После чего производится обход исследуемого изображения с перемещением окна на некоторый положительный шаг. По достижению конца изображения скользящее окно переносится в начало изображения и процесс поиска продолжается. Это может происходить, к примеру, до тех пор, пока размеры окна не совпадут с исследуемым изображением. То есть для каждого окна решается задача классификации – нужно решить чем является или не является объект в скользящем окне.

Также метод Виолы - Джонса базируется на следующих правилах:

1. Представление исследуемого изображение в интегральном виде. Это позволяет быстро производить вычисления.
2. Применение признаков Хаара[6], то есть признаков цифрового изображения, используемых в распознавании образов, с помощью которых происходит поиск нужного объекта
3. Использование бустинга(англ. boost - улучшение, усиление) для выбора более подходящих признаков для искомого объекта на данной части изображения.
4. Использование каскадов признаков для мгновенного отбрасывания окон, где объект не был найден

Рассмотрим каждый из принципов более подробно.

1.3.1 Интегральное представление изображения

Рассматриваемое представление является одним из основных этапов во многих методах распознавания объектов, например, в вейвлет-преобразованиях и SURF(Speeded Up Robust Features) методе. Интегральное представление изображения описывается матрицей, размерность которой совпадают с размерами исследуемого изображения. Формула для расчета элементов матрицы выглядит следующим образом

$$I(x,y) = \sum_{x' \le x, y' \le y} i(x', y'), (1.1)$$

где $i(x', y') -$ яркость пикселя исходного изображения.

Рекуррентный аналог формулы (1.1.)

$$I(x, y) = i(x, y) - I(x - 1, y - 1) + I(x, y - 1) + I(x - 1, y). (1.2)$$

Из (1.1) видно, что каждый элемент матрицы представляет собой сумму пикселей в прямоугольнике от $(0,0)$ до (x, y). То есть в каждом элементе хранится сумма яркостей всех пикселей, находящихся левее и выше данного элемента. Расчет матрицы производится за линейное время, пропорциональное числу пикселей на изображении. Таким образом интегральное представление позволяет рассчитывать суммарную яркость любого прямоугольника на изображении всего за четыре обращения к памяти. Примечательно и то, что время расчетов не зависит от площади прямоугольника. Для большей наглядности рассмотрим рисунок 2.

В верхнем прямоугольнике каждая ячейка представляет собой яркость пикселя исходного изображения. Нижний прямоугольник есть ничто иное, как интегральное представление изображения. Предположим, что нам нужно вычислить сумму пикселей S красного прямоугольника. Очевидно, что S внутри красного прямоугольника можно выразить через суммы и разности смежных прямоугольников в интегральном изображении по следующей формуле (1.3).

3	2	4	1
5	1	5	6
2	3	1	8
9	1	2	5
3	5	9	10
8	A 11	20	B 27
10	14	26	41
19	D 26	38	C 58

Рис. 2. Интегральное представление изображения.

$$S = A + C - B - D. \quad (1.3)$$

1.3.2 Признаки Хаара

Признаком f объекта a называется отображение $f: A \to D_f$, где D_f множество допустимых значений признака. Если задан набор признаков $f_{1\ldots}f_n$, то вектор $x = \left(f_1(a) \ldots f_n(a)\right)$ принято называть признаковым описанием объекта $a \in A$. Допускается не проводить различий между признаковым описанием и самим объектом. При этом множество $A = D_{f1} \times \ldots \times D_{fn}$ называют признаковым пространством. Для оригинального метода Виолы-Джонса используются прямоугольные признаки, которые называются примитивами Хаара.

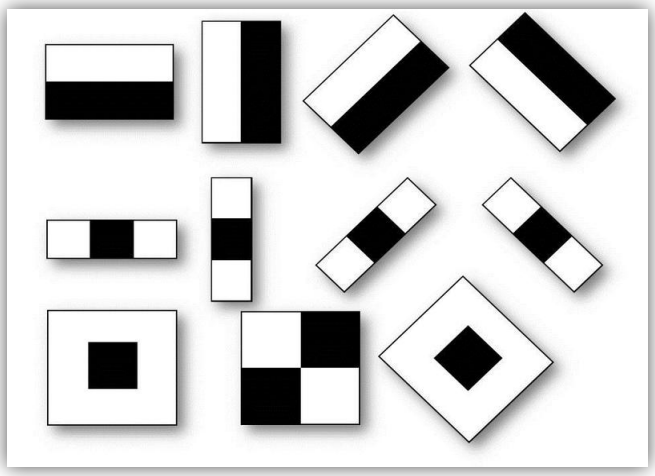


Рис. 3. Стандартные примитивы Хаара.

Но в расширенном методе, который в частности представлен в библиотеке OpenCV, используются и дополнительные примитивы.

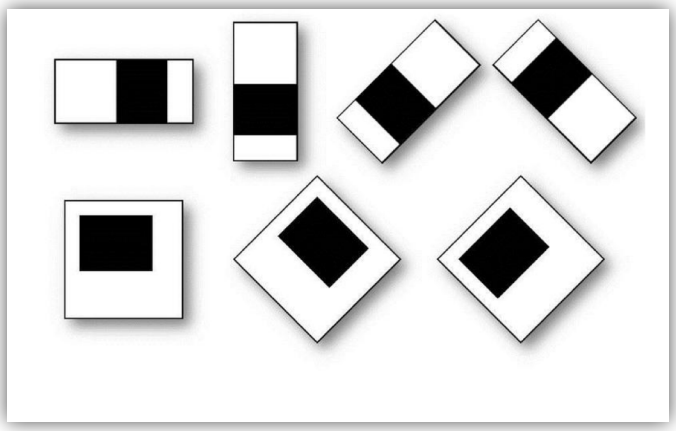


Рис. 4. Дополнительные примитивы Хаара.

Численным значением этого признака называется:

$$F = U - V,$$

где U – сумма значений яркостей точек, закрываемых светлой частью примитива Хаара, а V – сумма значений яркостей точек, закрываемых темной частью примитива Хаара. Для того, чтобы вычислить U и V прибегают к интегральному представлению исходного изображения. Рассмотрим примеры вычисления значения для следующих примитивов Хаара, основываясь на (1.3):

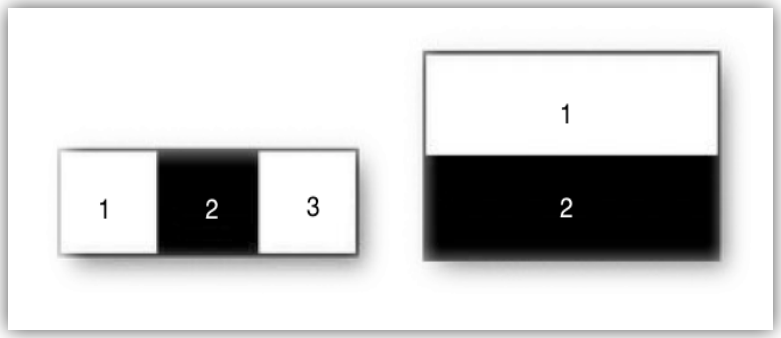


Рис. 5. Примитивы Хаара.

Для трех смежных прямоугольников $F = S_1 + S_3 - S_2$.

Для двух смежных прямоугольников $F = S_1 - S_2$,

где S_1, S_2, S_3 – суммы яркостей пикселей изображения, закрываемого прямоугольниками $1, 2, 3$ соответственно.

Оказывается, что для скользящего окна размером 24x24 пикселя существует 160000 комбинаций признаков Хаара. Очевидно, что для построения классификатора нам понадобятся не все. Для выбора нужных признаков будет использован алгоритм усиления классификаторов – AdaBoost[7].

1.3.3 AdaBoost

Бустинг (англ. boosting – улучшение) – это метод итеративного построения композиции алгоритмов машинного обучения, в котором каждый следующий алгоритм стремится компенсировать недостатки композиции всех

предыдущих алгоритмов. Бустинг является жадным алгоритмом, то есть алгоритм принимает локально оптимальные решения на каждом этапе, допуская, что конечное решение также окажется оптимальным. AdaBoost является алгоритмом адаптивного бустинга. Это означает, что каждый следующий классификатор строится по объектам, которые плохо классифицируются предыдущими классификаторами. В итоге AdaBoost строит строгий классификатор[2], представляющий собой линейную комбинацию взвешенных слабых классификаторов[3].

$$h(x) = sign\left(\sum_{j=1}^{M} \alpha_j h_j(x)\right), \qquad h_j(x) = \begin{cases} 1, s_j f_j < \theta_j \\ -1, s_j f_j \geq \theta_j \end{cases},$$

где θ_j – порог классификации, f_j – значение примитива данного объекта, $s_j \in \pm 1$– полярность классификатора, а x – классифицируемый объект.

Рассмотрим алгоритм AdaBoost:

Дано: $X^n = \{(x_1, y_1), .., (x_i, y_i)\}$ – обучающая выборка, где y_i – номер класса для образца x_i ($y_i \in \pm 1$);

1. Инициализируются веса $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$, где m и l – количество отрицательных и положительных образцов соответственно.
2. Для каждого $t = 1, \dots, T$:
   a. Нормируем веса $w_{t,i} = \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$
   b. Выбирается слабый классификатор с наименьшей ошибкой $\epsilon_t = min_{f,p,\theta} \sum_i^n w_i |h(x_i, f, p, \theta) - y_i|$
   c. Определяется классификатор $h_t(x) = h(x, f_t, p_t, \theta_t)$, где$f_t, p_t$ *и* θ_t значения, на которых достигается ϵ_t.

[2] Строгий классификатор – классификатор, допускающий мало ошибок классификации

[3] Слабый классификатор – классификатор, допускающий большое количество ошибок, не позволяя надежно определить класс исследуемого объекта

d. Обновляются веса $w_{t+1,i} = w_{t,i}\beta_t^{1-e_i}$, где функция потерь $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$, а $e_i = 0$ если объект x_i определен корректно и $e_i = 1$ в ином случае.

3. Получаем сильный классификатор

$$C(x) = \begin{cases} 1, \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2}\sum_{t=1}^{T} \alpha_t \\ 0, \sum_{t=1}^{T} \alpha_t h_t(x) < \frac{1}{2}\sum_{t=1}^{T} \alpha_t \end{cases},$$

где $\alpha_t = log\frac{1}{\beta_t}$.

Таким образом откидываются все примитивы, которые обнуляют значение функции классификатора.

Выделим достоинства и недостатки рассмотренного алгоритма.

Достоинства:

1. В некоторых задачах обобщающая способность[4] может улучшаться по мере увеличения числа базовых алгоритмов.
2. Легкость реализации
3. Время построения композиции практически полностью определяется временем обучения базовых алгоритмов, поэтому собственные накладные расходы бустинга невелики
4. Возможно идентифицировать объекты, являющиеся шумовыми выбросам(объектами с наибольшими весами)

Недостатки:

1. Если будет наблюдаться значительное количество шума в данных, то потребуется переобучение. Дело в том, что $\beta_t^{1-e_i}$ сильно увеличивает

[4] Говорят, что алгоритм обучения обладает способностью к обобщению(*обобщающей способностью*), если вероятность ошибки на тестовой выборке достаточно мала или хотя бы предсказуема, то есть не сильно отличается от ошибки на обучающей выборке.

веса объектов, на которых ошибаются многие базовые алгоритмы. Однако весьма вероятно что эти объекты окажутся шумовыми выбросами. Как результат, AdaBoost начинает настраиваться на шум, что заставляет переобучаться. Эта проблема разрешается путём удаления выбросов или применения менее “агрессивных” функций потерь. В частности, применяется алгоритм GentleBoost;

2. Для AdaBoost требуется достаточно длинная обучающая выборка.
3. Жадная стратегия AdaBoost может приводить к построению неоптимального набора базовых алгоритмов. Для улучшения композиции следует периодически возвращаться к ранее построенным алгоритмам и переобучать их.
4. Бустинг может приводить к построению композиций, состоящих из сотен алгоритмов. Такие огромные композиции требуют больших объёмов памяти для хранения базовых алгоритмов и существенных временных затрат на вычисление классификаций.

Так или иначе на сегодня подход усиления слабых классификаторов является одним из наиболее эффективных методов классификации ввиду своей высокой скорости, эффективности работы и относительной простоты реализации.

1.3.4 Использование каскадов

Для того, чтобы быстро отсеивать окна с нехарактерными признаками для искомого объекта используется каскадная модель сильных классификаторов, Эта модель представляет собой дерево принятия решений, то есть такое дерево, в листьях которого находятся значения целевой функции[5], а в остальных узлах — атрибуты, по которым различаются случаи. На рисунке 6

[5] Целевая функция – вещественная или целочисленная функция нескольких переменных, подлежащая оптимизации (минимизации или максимизации) в целях решения некоторой оптимизационной задачи.

представлен пример дерева принятия решений:

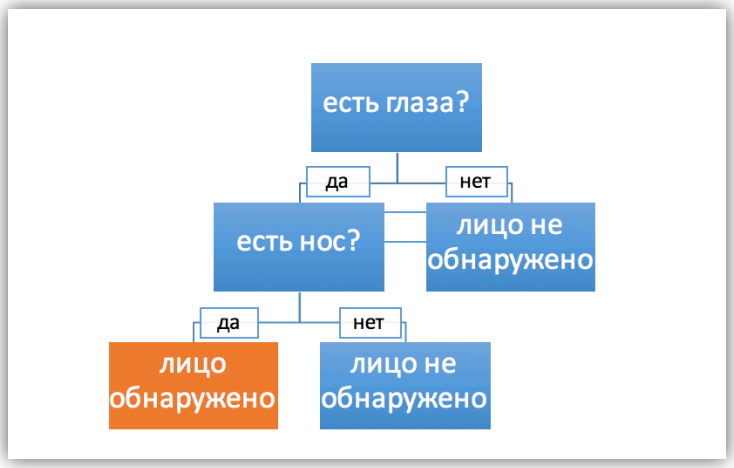


Рис. 6. Дерево принятия решений.

Безусловными достоинствами таких деревьев – это наглядность и легкость работы с ними.

По сути каждый узел каскадной модели сильных классификаторов есть классификатор, построенный на определенном количестве примитивов. Причем, чем ближе узел находится к корню дерева, тем из меньшего количества примитивов он состоит. Таким образом, расположение узла в дереве влияет на количество вычислительных операций для принятия решения. Это позволяет детектировать почти все искомые образы и отклонять окна, в которых нужный образ не найден. Скользящее окно, которое проходит через все уровни каскада определяется, как искомый объект. Пример каскадной модели изображен на рисунке 7. Отметим, что сложность обучения таких каскадов равна $O(xyz)$, где x – количество уровней, y – примеров, z – признаков.

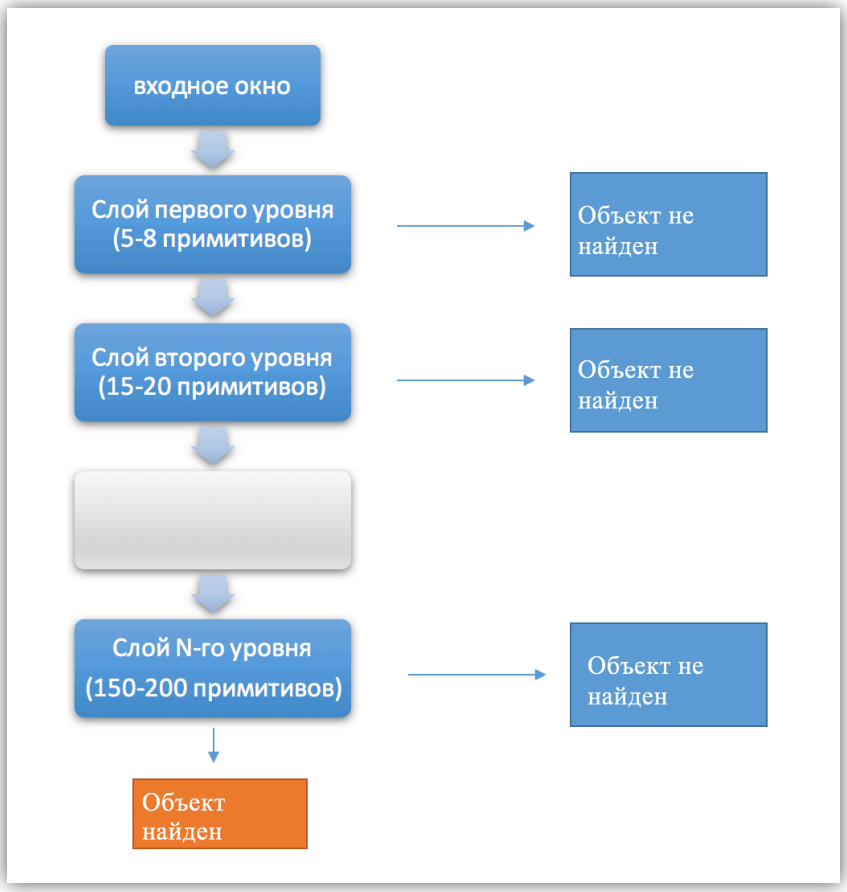


Рис. 7. Каскад классификаторов.

1.4 Обзор современных мобильных операционных систем

На сегодняшний день наиболее популярными мобильными операционными системами являются: Android, iOS и Windows Phone[8]. Выделим некоторые особенности каждой из них. Будучи основанной на ядре Linux, Android имеет значительное преимущество перед iOS и Windows Phone – это открытый исходный код ОС. Это дает разработчикам практически неограниченные возможности для написания программ под эту платформу. В то же время пользователям этой операционной системы позволено устанавливать программы из любых источников, что иногда приводит к поломке Android устройств, так как часть из таких программ может содержать вредоносный код. Android является самой популярной операционной системой, занимая 83% рынка мобильных ОС. Что касается Windows Phone, то это сравнительно молодая операционная система, представленная компанией

Microsoft 11 октября 2010 года. К сожалению, интересные дизайнерские решения, стабильность и простота использования этой платформы не улучшает позиций Microsoft на рынке мобильных операционных систем. Windows Phone все еще страдает от недостатка некоторых приложений и ее доля на рынке составляет всего 3%. Второй по популярности операционной системой является iOS от компании Apple. В отличие от Windows Phone и Android, iOS выпускается только для устройств, производимых фирмой Apple. Как следствие одним из главных преимуществ этой операционной системы становится отсутствие фрагментации платформы[6]. Это дает разработчику возможность написание приложения для нескольких iOS устройств одновременно, в то время, как создание приложения для Android порождает ряд трудностей, ведь приходится учитывать форм-фактор и версию операционной системы каждого Android устройства. Фрагментация становится еще более критичной для приложений, работающих с камерой, так как часть смартфонов на базе Android и Windows Phone имеют некачественные камеры, чего нельзя сказать о смартфонах на базе iOS.

1.5 Обзор средств разработки

При разработке приложения для Windows Phone(WP) требуется наличие компьютера с операционной системой Windows и интегрированная среда разработки(IDE) – Visual Studio. Основным языком программирования для WP является C#. Приложения для Android традиционно создаются с использованием языка Java в таких IDE, как Android Studio и Eclipse, которые доступны для Linux, Windows и OS X. В настоящее время компания Google прекратила поддержку плагина Android Development Tools (ADT) для Eclipse,

[6] Фрагментация платформы — ситуация, когда у какой-то вычислительной платформы становится настолько много моделей аппаратуры и версий ОС, что становится практически невозможным написать программу, хорошо работающую на всех устройствах, созданных на базе данной вычислительной платформы.

сделав Android Studio основной средой создания Android приложений. Разработка iOS приложений ведется исключительно на операционной системе OS X, в специальных IDE: Xcode от Apple или AppCode от Jetbrains. Когда речь идет о написании кода, то AppCode является лучшим из двух вариантов. Процесс автоматического изменения внутренней структуры программы с целью облегчения понимания её работы в AppCode значительно превосходит Xcode. IDE от Jetbrains способна обнаруживать “плохие” куски кода и предлагать различные альтернативы, улучшающие код. К сожалению, в отличие от Xcode AppCode не поддерживает storyboards[7], что становится весьма критичным при разработке приложения с несколькими экранами. До недавнего времени основным языком программирования iOS приложений был Objective-C[8], однако 2 июня 2014 года на конференции WWDC[9] был представлен открытый мультипарадигменный объектно-ориентированный язык программирования – Swift. Обозначим основные преимущества нового языка от Apple над Objective-C:

1. Безопасность Swift:

   Один из интересных моментов в Objective-C – способ обработки указателей: может показаться, что приложение не “упадет”, если будет вызван метод с неинициализированной переменной указателя. Но на самом деле выражение становится невыполнимым. Такого рода

[7] Storyboards – визуальное представление пользовательского интерфейса iOS приложения. Помимо основного контента на каждом экране storyboards показывает связи между этими экранами.

[8] Objective-C — компилируемый объектно-ориентированный язык программирования, используемый корпорацией Apple, построенный на основе языка C и парадигм Smalltalk. Язык Objective-C является надмножеством языка C, поэтому код C полностью понятен компилятору Objective-C.

[9] Apple Worldwide Developers Conference, обычно сокращённо WWDC — всемирная конференция разработчиков на платформе Apple. Проводится ежегодно в Калифорнии, США.

поведение является серьезной проблемой для программистов, пытающихся найти и исправить случайные сбои или остановить отклоняющееся от нормы поведение.

В Swift опциональные типы дают возможность существования в коде nil опционального значения. Если вызовется метод с nil, то создастся ошибка компилятора. Это позволяет программистам писать код более уверено, так как проблемы могут быть устранены при уже написанном коде.

2. Поддержка кода:

   Как известно стандарт оформления кода C требует от программистов поддержки 2 файлов для создания приложения. Это требование распространяется и на Objective-C. Swift отменяет это требование. В результате, повторяющаяся задача разделения оглавления (файл заголовка) от тела (файл реализации) становится делом прошлого. Swift сочетает в себе заголовок Objective-C (.h) и файлы реализации (.m) в одном файле кода (.swift).

3. Управление памятью Swift:

   В Objective-C программисту приходится брать на себя управление памятью при работе с Core Graphics APIs, и другими более старыми API, доступных на iOS. Поэтому в коде неопытных программистов часто наблюдаются утечки памяти. В Swift такие утечки невозможны, так как поддержка автоматического подсчета ссылок (ARC) в Swift является полной по процедурным и объектно-ориентированным путям кода.

4. Скорость Swift:

   Тесты производительности кода указывают на то, что Swift быстрее чем Objective-C[9].

Глава II. Практическая часть

Ввиду наличия программно-аппаратных средств, а также желания изучить Swift было решено использовать операционную систему iOS, как платформу для разработки мобильного приложения. В качестве основной IDE была выбрана Xcode, так как в отличие от AppCode является бесплатной средой разработки и поддерживается компанией-производителем iOS устройств.

Для разработки алгоритма распознавания в работе используется библиотека компьютерного зрения – OpenCV[10]. Данная библиотека подчиняется условиям лицензии BSD, то есть может свободно использоваться в коммерческих и академических целях. Будучи реализованной на языке C/C++, OpenCV также разрабатывается для Python, Java, Ruby и Matlab. К сожалению, на сегодняшний день не существует реализации OpenCV для Swift. Более того, использование C/C++ напрямую в Swift проекте – невозможно[11], но совместимость Swift с Objective-C позволяет создать проект, содержащий код, написанный на обоих языках. В свою очередь код на C++ встраивается в Objective-C проекты. Традиционно такое встраивание достигается с помощью использования диалекта Objective-C++. Итак, идея использования OpenCV в Swift проекте выглядит следующим образом:

1. Создать Swift проект в Xcode
2. Добавить в проект Objective-C++ класс.
3. Добавить в файл реализации Objective-C++ класса заголовочный файл, в котором описаны C++ функции, используемые в приложении.
4. Так как разрабатываемое приложение будет работать с изображениями, необходимо добавить в проект категорию, которая позволит

установить изоморфизм между классом библиотеки OpenCV – Mat[11] и классом Swift – UIImage[12].

5. Создать мост(заголовочный файл), позволяющий использовать функции Objective-C в Swift.
6. Добавить заголовочный файл Objective-C++ класса в мост.
7. Использовать функции из Objective-C++ класса в файлах Swift

После выполнения перечисленных действий появится возможность использования фреймворка OpenCV в Swift проекте.

2.1 Реализация детектора мимических движения

Очевидно, что для минимизации ошибки распознавания мимических движений, для начала следует решить задачу детектирования лица. Для этого нам понадобится каскадный классификатор Хаара. Для работы с данным классификатором в OpenCV используются объекты класса `CascadeClassifier`[12]. Конструктор этого класса принимает на вход путь к файлу, содержащего натренированную модель. В составе установленного пакета OpenCV присутствует натренированная модель для детектирования лиц, что позволит сэкономить время на обучении классификатора.

Для нахождения объектов разного масштаба на изображении используется функция `detectMultiScale`. Данная функция имеет важное значение в разрабатываемом приложении, поэтому разберем входные параметры этой функции.

```
void detectMultiScale(const Mat& image, vector<Rect>& objects,
double scaleFactor=1.1, int minNeighbors=3,  int flags=0, Size
minSize=Size());
```

[11] Класс Mat представляет собой численный n-мерный массив, который может быть использован для хранения вещественных или комплекснозначных векторов и матриц, черно-белых или цветных изображений, векторных полей, а также тензоров и гистограмм.

[12] UIImage – класс Swift для хранения изображений

Параметр `image` – это исследуемое изображение, преобразованное в черно-белый цвет. В `objects` записываются прямоугольники, окаймляющие распознанные лица. Третьем параметром является коэффициент масштабирования – `scaleFactor`, определяющий во сколько раз будет уменьшено изображение на каждом шаге. То есть в отличие от оригинального метода Виолы-Джонса, где происходит увеличение скользящего окна, функция `detectMultiScale` уменьшает исследуемое изображение при каждом обходе, создавая при этом пирамиду изображений[13]. Параметр `minNeighbors` служит для того, чтобы отсекать одиночные срабатывания детектора, которые с большой долей вероятности являются ложными, а близкие по расположению срабатывания объединять в единую сущность. Что касается параметра `flags`, то в текущей реализации классификатора он не используется и присутствует только для совместимости с интерфейсом устаревшей функции `cvHaarDetectObjects`. Параметр `minSize` определяет минимальный размер искомого объекта.

После того, как лицо было распознано, необходимо обрезать исходное изображение до размеров окаймляющего прямоугольника, тем самым получить новое изображение, содержащее только лицо. На полученном изображении необходимо определить два мимических движения. Первое – это наличие улыбки, а второе – подмигивание. Очевидно, что рот человека может находятся только в нижней части лица, а глаза только в верхней, поэтому разделим полученное изображения пополам, как показано на рисунке 8.

Рис. 8. Разделенное лицо.

В комплект фреймворка OpenCV входит натренированная модель для детектирования улыбок, поэтому остается лишь вызвать функцию `detectMultiScale` для нижней части лица, тем самым определив улыбается ли человек.

Для того, чтобы распознать подмигивание, сначала следует найти пару глаз на верхней части лица. Далее, определить состояние каждого глаза, то есть открыт он или закрыт. Для этого полученное изображение с обоими глазами необходимо разделить пополам и обрабатывать каждый глаз отдельно. Если один глаз распознается, как открытый, а на другой половине изображения детектировать открытый глаз не удается, то делается вывод о том, что человек подмигивает. К сожалению, в OpenCV отсутствует модель для детектирования открытого глаза, поэтому для того, чтобы избежать обучения классификатора, который требует большой обучающей выборки, было решено воспользоваться преобразованием Хафа для нахождения открытого глаза. В OpenCV это преобразование реализуется с помощью функции `HoughCircles`. возвращающая количество найденных окружностей – `circles`.

Идея применения `HoughCircles` заключалась в преобразовании изображения так, чтобы радужная оболочка глаза выглядела как замкнутая окружность (если окружность найдена, то глаз считается открытым, иначе закрытым). Для реализация данного подхода изображение с глазом было переведено в черно-белое цветовое пространство(`cvtColor`), после чего дополнительно снижен уровень шума (`GaussianBlur`) и применен детектор границ Кэнни(`Canny`).

На следующем рисунке представлены этапы преобразования изображения при различных условиях освещенности:

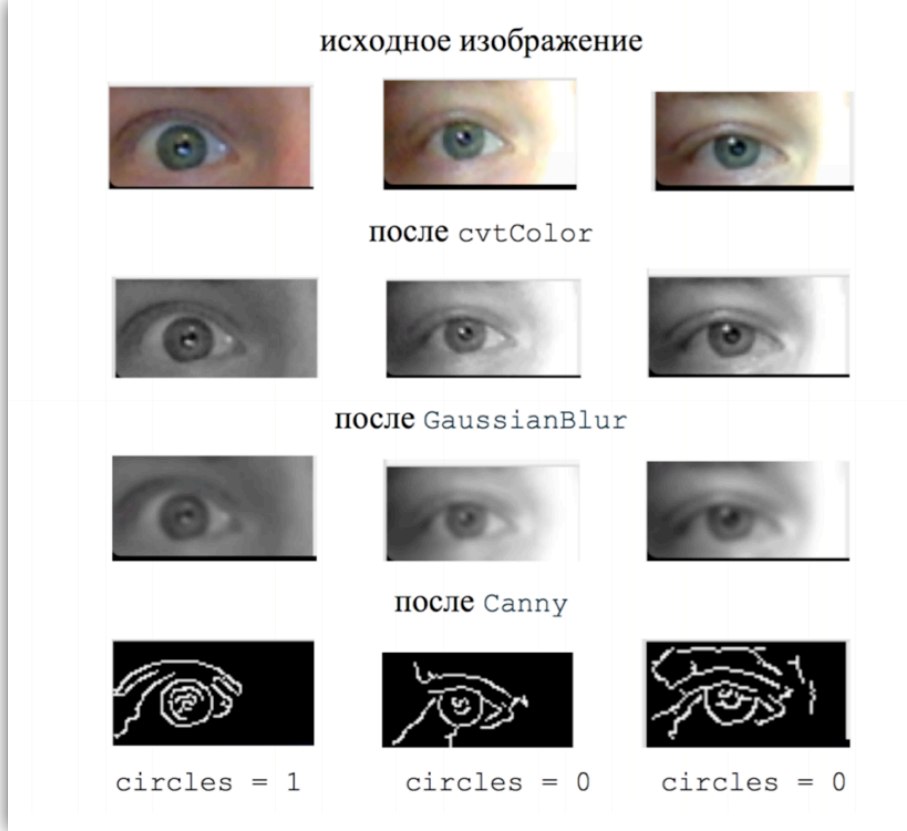


Рис. 9. Преобразование изображения глаза.

Как видно из рисунка 9, освещенность лица и “мера открытости” глаза влияют на границы, которые выделяет алгоритм Кэнни. Как итог, функция `HoughCircles` не способна вернуть требуемое количество окружностей – `circles`, препятствуя детектировать подмигивание. Попытки подбора универсальных параметров для детектора границ Кэнни не принесли желаемого результата(более подробное описание результатов будет приведено в параграфе 2.1.2), поэтому для распознавания открытого глаза было решено обучить каскадный классификатор, который устойчив к изменениям освещенности как следует из главы I.

2.1.1 Обучение классификатора

Для обучения классификатора был использован набор данных CEW(Closed Eyes In The Wild) [14]. Из этого набора было выделено 1423 положительных примера размером 24x24 пикселя с открытым глазом. В качестве отрицательных примеров[15] из этого же набора данных было

нарезано 1478 фотографий, содержащие все части лица кроме открытого глаза. Перед началом обучения необходимо создать две папки. “Good” – папка с положительными примерами, “Bad” – с отрицательными.

Рис. 10. Положительные примеры

Рис. 11. Отрицательные примеры

Также понадобятся текстовые файлы, описывающие используемые изображения: “good.dat” и “bad.dat”. Файл “good.dat” выглядит следующим образом:

```
/path/1.jpg  1  0 0 23 23
/path/2.jpg  1  0 0 23 23
```

Где `/path/1.jpg` – полный путь к изображению. `1` – количество положительных объектов на изображении. `0 0 23 23` – координаты прямоугольника, содержащего объект, причем первые две координаты определяют верхний левый угол прямоугольника, а вторые – его длину и высоту. Файл “bad.dat” имеет более простую структуру и содержит только путь к отрицательному изображению.

После того, как все данные подготовлены, положительные изображения приводятся к общему формату. Это процедура проделывается при помощи утилиты `opencv_createsamples`, входящей в состав пакета OpenCV. На вход этой утилиты подается файл описания положительных изображений, файл, в

который будет сохранена приведённая к общему формату база положительных изображений, и размер шаблона, отражающий пропорции искомого объекта.

Для создания итогового каскада использовалась утилита `opencv_traincascade`. Рассмотрим основные ключи этой утилиты:

`-data` – путь папки, в которую сохраняются итоговый каскад.

`-vec` – файл, полученный после применения утилиты `opencv_createsamples`.

`-bg` – путь файла описания отрицательных примеров

`-numStages` – количество уровней каскада, которые `opencv_traincascade` будет обучать.

`-minhitrate` – коэффициент, определяющий качество обучения.

`-maxFalseAlarmRate` – уровень ложной тревоги

`-numPos` – количество позитивных примеров. Здесь важно отметить тот факт, что необходимо указывать число, меньшее количества положительных примеров. Дело в том, что чем ниже коэффициент "`minhitrate`", тем больше примеров будут считаться непригодными.

`-numNeg` – количество отрицательных примеров.

`-w -h` – ключи, служащие для определения размеров шаблона.

`-mode` – комплект примитивов Хаара. Доступные значения: `BASIC`, `CORE`, `ALL`.

`-precalcValBufSize -precalcIdxBufSize` – выделяемая под процесс память.

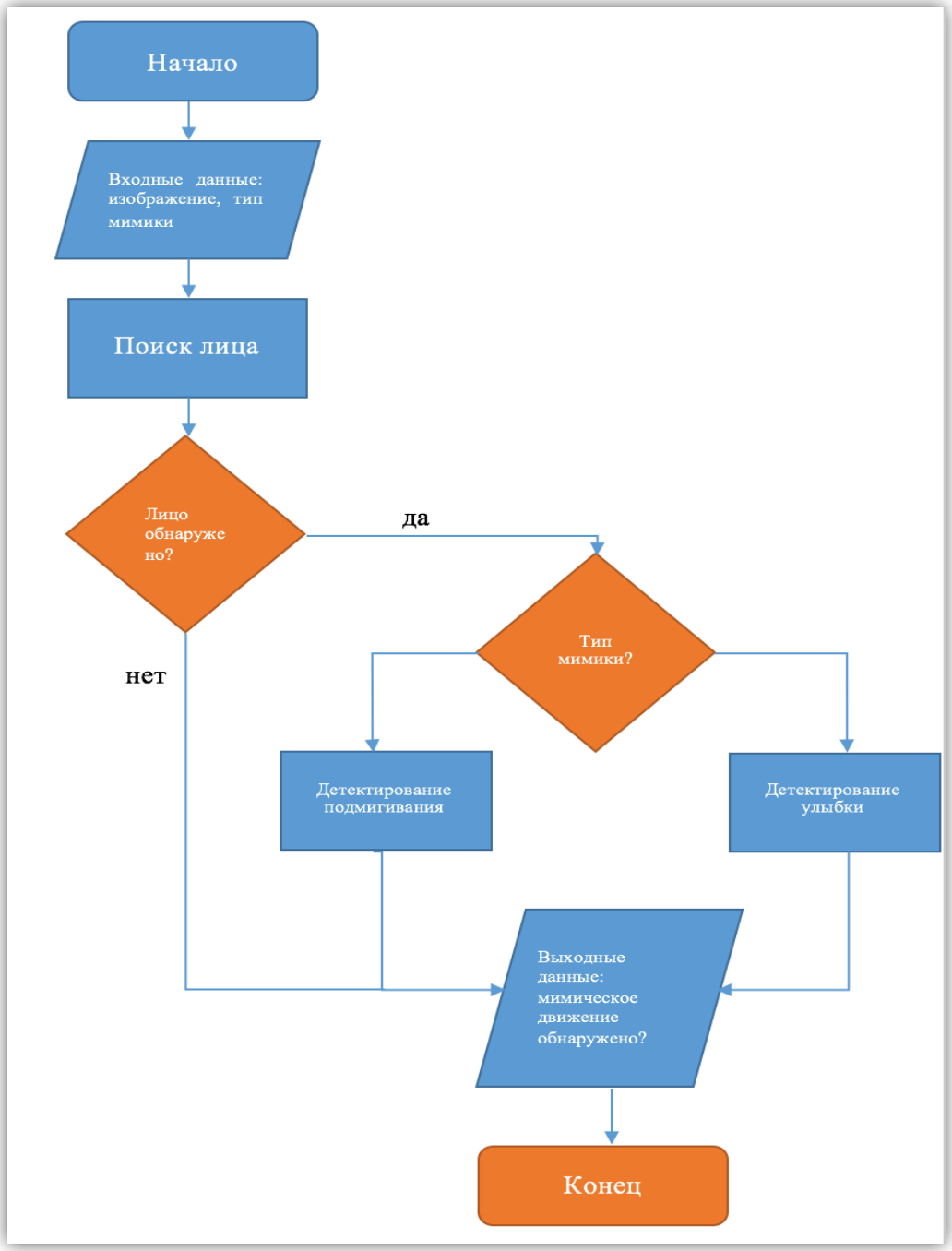
Обучение происходило на ноутбуке с операционной системой OS X El Capitan, с процессором Intel Core i5 2,6ГГц и с 8ГБ ОЗУ. Утилита `opencv_traincascade` запускалась со следующими параметрами:

```
./opencv_traincascade -data /Volumes/Macintosh\
HD/Users/Albert/Desktop/Diploma/ -vec samples.vec -bg bad.dat -
numStages 24 -minhitrate 0.999 -maxFalseAlarmRate 0.4 -numPos
1138 -numNeg 1478 -w 24 -h 24 -mode ALL -precalcValBufSize 1024
-precalcIdxBufSize 1024
```

Процесс обучения занял 1 час 38 минут и 42 секунды.

2.1.2 Реализация алгоритма распознавания

Для распознавания мимических движений на языке C++ были реализованы функции `detectBlink` и `detectSmile`, отвечающие за распознавание подмигивания и улыбки соответственно. Ниже приведена общая схема работы этих функции.



Так как задача итогового решения – это создание селфи, то функции `detectBlink` и `detectSmile` отбрасывают изображения, в которых обнаружено больше одного лица.

2.1.3 Тестирование детектора мимических движений

Для тестирования алгоритмов распознавания с камеры смартфона iPhone 5s было получено 500 кадров: 250 для детектирования улыбки и 250 для детектирования подмигивания. Первая половина состояла из фотографий лиц, улыбающихся и не улыбающихся людей. Вторая половина включала в себя три типа фотографий: закрыт один глаз, открыты оба глаза, закрыты оба глаза. Важно отметить, что тестирование производилось в различных условиях освещенности. Из сформированной тестовой выборки для распознавания подмигивания было выделено 145 фотографий, на которых лицо освещено равномерно. Остальные 105 фотографий были классифицированы, как плохо освещенные. Для оценки качества алгоритма использовалась метрика $Accuracy$ или аккуратность:

$$Accuracy = \frac{P}{N},$$

где P – количество кадров, по которым классификатор принял правильные решение, а N – количество тестируемых кадров.

Рис. 12. Пример равномерно освещенного лица

Рис. 13. Пример неравномерного освещенного лица

Для оценки качества распознавания подмигивания, были протестированы две реализации функции `detectBlink`: реализация, основанная на методе Виолы-Джонса(реализация 1) и реализация, основанная на применении детектора границ Кэнни и преобразовании Хафа(реализация 2).

Таблица 1. Показатели `detectBlink` на равномерно освещенных фотографиях

`detectBlink`	P_b	$Accuracy_b$	Скорость обработки
реализация 1	138	~0.95	100 мс
реализация 2	69	~0.48	40 мс

Таблица 2. Показатели `detectBlink` на неравномерно освещенных фотографиях

`detectBlink`	P_b	$Accuracy_b$	Скорость обработки
реализация 1	84	0.8	100 мс
реализация 2	15	~0.14	40 мс

Как видно из таблицы 1 и таблицы 2, несмотря на свою скорость работы,

реализация функции `detectBlink`, основанная на детекторе границ Кэнни и преобразовании Хафа не могла быть использована в разрабатываемом приложении, в силу своего низкого показателя аккуратности. Для реализации 1: $Accuracy_{b1} = 0.888$, при $P_{b1} = 222$. Для реализации 2: $Accuracy_{b2} = 0.336$, при $P_{b2} = 84$. Поэтому в качестве основы для распознавания подмигивания была выбрана реализация 1.

При распознавании улыбки результаты оказались следующими: $Accuracy_s = 0.972$, при $P_s = 243$ и скорости обработки $V_s = 80$ мс. $Accuracy_{b1} < Accuracy_s$, так как классификатор улыбок, входящий во фреймворк OpenCV, был обучен на более качественной выборке большего размера, нежели классификатор, обученный в рамках данной работы.

2.2 Реализация функциональной части приложения

Основным фреймворком для создания iOS приложений является Cocoa Touch, который следует шаблону проектирования Model-View-Controller. Cocoa Touch предоставляет уровень абстракции[13] для операционной системы iOS и основана на классах фреймворка Cocoa, используемого в OS X. Cocoa Touch включает в себя несколько важных библиотек, часть из которых была задействована в данной работе. Перейдем к рассмотрению используемых в приложении библиотек:

1. `Foundation framework` – библиотека, определяющая базовый уровень функциональности. `Foundation framework` включает в себя все необходимые примитивы для создания iOS/OS X приложений.
2. `UIKit framework` – библиотека, которая обеспечивают всю инфраструктуру для управления пользовательским интерфейсом приложения.

[13] В вычислительной технике уровнем абстракции называется способ сокрытия деталей реализации определенного набора функциональных возможностей.

3. `AVFoundation framework` – библиотека, предоставляющая все необходимые API для работы с аудио и видео устройствами на iOS и OS X. Эта библиотека включает в себя класс `AVCaptureVideoDataOutput`, основной задачей которого является получение данных с камеры смартфона.
4. `Social framework`[16] – библиотека, позволяющая интегрировать iOS приложение с такими социальными сетями, как Twitter, Facebook и Weibo.
5. `SwiftyVK`[17] – сторонняя библиотека, позволяющая приложению взаимодействовать с социальной сетью ВКонтакте.

При проектировании графического интерфейса приложения было решено использовать три экрана, как показано на рисунке 14. На первом экране воспроизводится видео, полученное с фронтальной камеры смартфона. После того, как человек подмигивает, запускается таймер. По истечению 5 секунд камера фотографирует человека и приложение переходит ко второму экрану, где отображается сделанный фотоснимок. Если пользователю нравится этот снимок, то ему необходимо улыбнуться для того, чтобы перейти к третьему экрану. Третий экран предоставляет пользователю возможность загрузки фотографии в социальные сети.

По сути, каждый экран приложения определяет этап взаимодействия пользователя и программы. Задача первого и второго этапа схожи – необходимо получить данные с камеры и распознать нужные мимические движения. Третий же отвечает за отправку фотографий в различные социальные сети. Опишем архитектуру этих этапов более подробно.

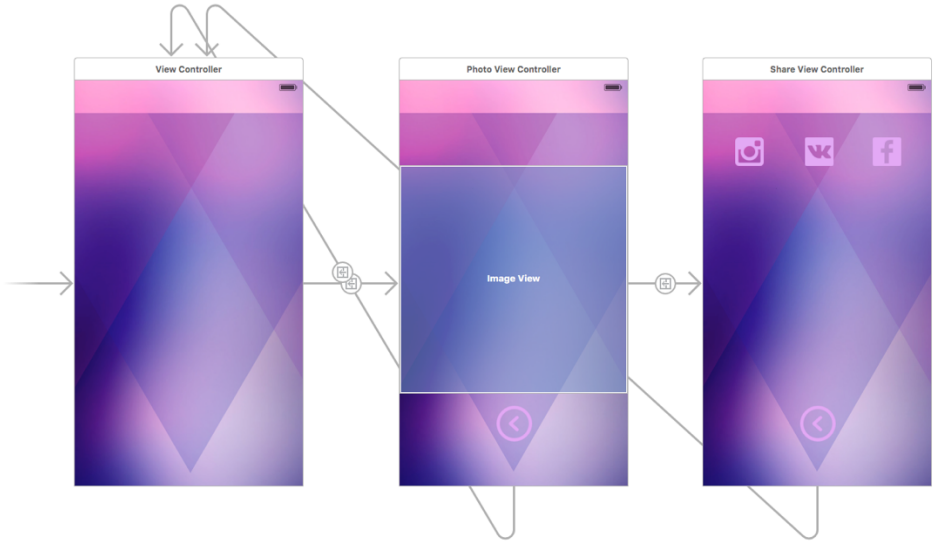


Рис. 14. Графический интерфейс приложения.

2.2.1 Этап 1: создание фотографии

После инициализации сессии, которая отвечает за захват видео с камеры, следовало вывести получаемое видео на экран. Для этого был добавлен дополнительный слой поверх основного фона приложения.

Основная задача, которая решается на первом экране – определить подмигивает ли пользователь приложению. Очевидно, что видео, получаемое с камеры следует обрабатывать как последовательность кадров[18]. Для этой цели был создан объект класса `AVCaptureVideoDataOutput` – `videoCaptureOutput`. Созданному объекту был назначен делегат[14], который позволяет записывать данные, получаемые с камеры смартфона в некоторый буфер. Важно отметить, что захват видео происходит при максимальной кадровой частоте фронтальной камеры. Для iPhone 5s эта частота равна 60 кадрам в секунду. Очевидно, что для мобильного устройства обрабатывать

[14] Делегирование - это шаблон, который позволяет классу или структуре передавать (или делегировать) некоторую ответственность экземпляру другого типа. Этот шаблон реализуется определением протокола (интерфейса), который инкапсулирует делегируемые полномочия, таким образом, что соответствующий протоколу тип (делегат) гарантировано получит функциональность, которая была ему делегирована.

кадры с такой частотой затруднительно(средняя скорость обработки изображения на iPhone 5s составляет 12.5 кадров в секунду), поэтому `videoCaptureOutput` отбрасывает кадры, которые приходят с камеры в момент обработки заполненного буфера, не создавая очередей. Чтобы преобразовать заполненный буфер в объект класса `UIImage`, было написано расширение[15] этого класса.

После того, как изображение готово к обработке, через реализованный ранее мост должна быть вызвана функция `detectBlink`, в которой происходит инициализация объектов класса `CascadeClassifier`. Но дело в том, что в iOS не предоставляется возможности получения перманентного пути к файлам проекта(при каждом запуске приложения файлы с натренированными каскадами имеют "новое" местонахождение). Поэтому, помимо полученного изображения, `detectBlink` принимает на вход пути к каскадным моделям лица, пары глаз и открытого глаза. Для получения пути к этим моделям используется Swift функция – `pathForResourse`, возвращающая объекты типа String(Swift). Эти объекты преобразуются в объекты типа string(C++), после чего происходит вызов функции `detectBlink`. Если `detectBlink` выносит вердикт – "подмигивание зафиксировано", то запускается таймер, по истечению которого приложение создает фотографию и переходит ко второму экрану.

2.2.2 Этап 2: оценка фотографии

На данном этапе работы приложения полученный снимок передается второму экрану. После чего посредством функции `detectSmile` происходит обработка кадров получаемых с камеры. Таким образом, на данном экране пользователю предоставляется две возможности:

[15] Расширения необходимы для того, чтобы добавлять новые функциональные возможности в существующие классы, структуры и перечисления, не имея доступа к исходному коду.

1. Улыбнуться и перейти к экрану отправки фотографии в социальные сети.
2. Вернуться на предыдущий экран захвата изображений, нажав соответствующую кнопку, расположенную ниже полученного снимка.

2.2.3 Этап 3: отправка фотографии в социальные сети

На третий экран были добавлены три кнопки, каждой из которых была присвоена некоторая функция, отвечающая за отправку полученного снимка в следующие социальные сети: Facebook, Instagram и ВКонтакте. Основными причинами выбора этих сетей является их популярность и то, что для каждой из них существует своя специфическая методика загрузки фотографий через iPhone. Начнем с разбора загрузки фотографии в Facebook.

Существует несколько вариантов загрузки фотографий в Facebook с помощью iOS устройства. Первый и в то же время наиболее простой способ – это передача изображения в приложение Facebook(данный способ будет рассмотрен на примере с Instagram). Второй способ – использование фреймворка Social от Apple. Третий способ подразумевает использование Facebook SDK. Такой подход теряет свою актуальность для таких задач, как загрузка текста, фотографий и видео ввиду большей сложности, поэтому был задействован фреймворк Social.

После нажатия на кнопку для отправки фотографии в Facebook, проверяется доступность учетной записи Facebook пользователя iOS устройства. Далее появляется окно(рисунок 15) с фотографией и хэштегом[16] приложения –“#Blink”. Дополнительно можно указать место публикации и добавить текст, описывающий фотографию.

[16] Хэштег или хештег (метка) (англ. hashtag от hash – символ «решётка» + tag – тэг) — слово или фраза, которым предшествует символ #. Пользователи используют хэштеги для объединения группы сообщений по теме или типу.

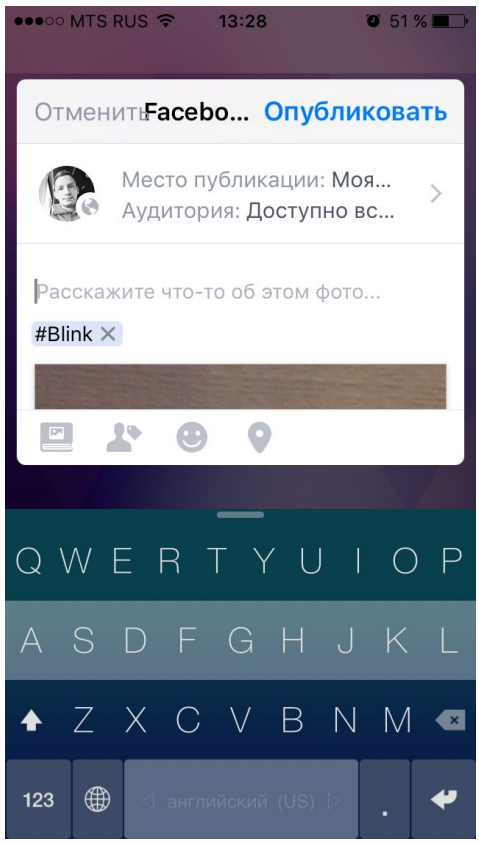


Рис. 15. Окно отправки фотографии в Facebook.

Перейдем к описанию загрузки фотографии в социальную сеть Instagram. К сожалению, Instagram не предоставляет API для загрузки контента из сторонних приложений, поэтому была использована техника под названием iPhone Hooks[19]. Суть этой техники заключается в передаче фотографии из своего приложения в поток публикаций Instagram, где можно накладывать различные фильтры на фотографию. Таким образом, в функции, которая отвечает за загрузку фотографии в рассматриваемую социальную сеть, необходимо было проверить наличие приложения Instagram на iOS устройстве. Если приложение установлено, то предлагается возможность передачи снимка в Instagram и применения фильтров, как показано на рисунке 16.

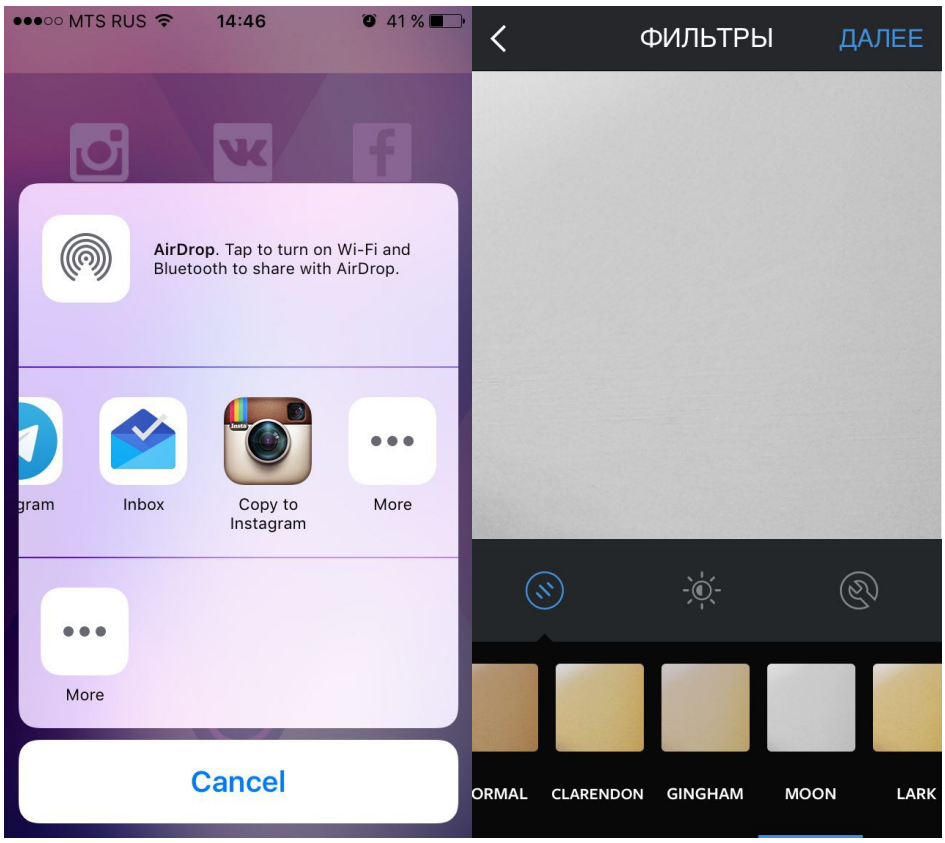


Рис. 16. Иллюстрация работы iPhone Hooks.

Для социальной сети ВКонтакте было решено использовать библиотеку SwiftyVK, которая реализована на языке Swift. Вместо скачивания библиотеки из репозитория и копирования ее в проект вручную, SwiftyVK была установлена при помощи средства управления зависимостями библиотек – CocoaPods. Также было необходимо зарегистрировать приложение на сайте ВКонаткте для получения ID приложения, который используется при отправки запросов к серверам этой социальной сети. Таким образом после нажатия кнопки с логотипом ВКонтакте появляется окно для авторизации, причем в случае наличия официального приложения от ВКонтакте(VK) на используемом iOS устройстве будет предложена авторизация через VK. Если официальное приложение от ВКонтакте не установлено, то окно авторизации появится поверх третьего экрана приложения, как показано на рисунке 17.

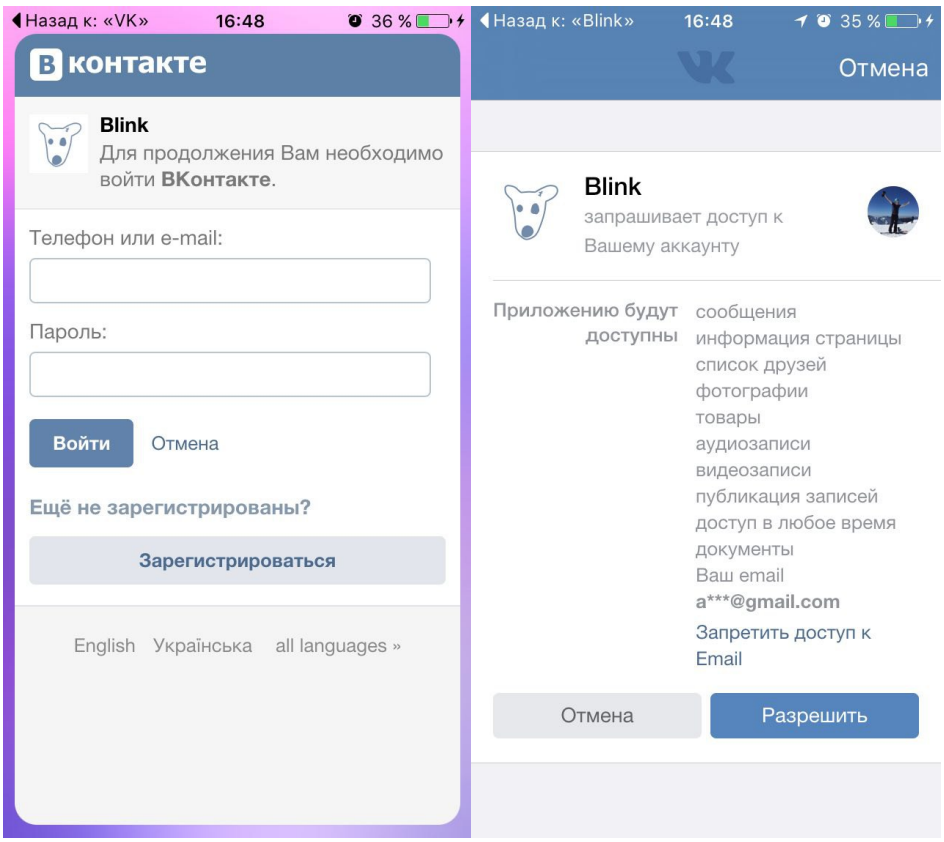


Рис. 17. Иллюстрация работы SwiftyVK.

Выводы

В ходе работы были получены следующие результаты:

1. Протестированы преобразование Хафа и детектор границ Кэнни для определения открытого глаза. Освещенность лица пользователя приложения оказалась критичным фактором при применении этих методов, поэтому от такого подхода было решено отказаться.
2. Сформирована обучающая выборка и обучен классификатор для детектирования открытого глаза средствами фреймворка OpenCV.
3. На основе метода Виолы-Джонса разработан алгоритм детектирования подмигивания и улыбки.
4. Реализовано приложение для iOS, позволяющее распознавать мимические движения и загружать фотографии в социальные сети. Данное приложение доступно на веб-сервисе GitHub[20].

Заключение

В рамках данной выпускной квалификационной работы был разобран и изучен процесс создания iOS приложения в среде Xcode. Было разработано приложение, способное распознавать такие мимические движения как подмигивание и улыбка, а также был реализован функционал работы с социальными сетями. Была изучена возможность интеграции языка C++ в Swift проект. В перспективе в данное приложение планируется добавить опцию съемки видео и распознавание дополнительных мимических движений. После чего разработанное приложение будет выложено в магазин приложений от Apple – AppStore на бесплатной основе.

Список литературы

1. Selfie infographic (2001): http://techinfographics.com/selfie-infographic-selfiegraphic-facts-and-statistics/
2. Canny J. F. A computational approach to edge detection // Pattern Analysis and Machine Intelligence, IEEE Transactions on. 1986. №6. P. 679-698.
3. Duda, R. O., P. E. Hart Use of the Hough Transformation to Detect Lines and Curves in Pictures // Comm. ACM, 1972, Vol. 15, P. 11–15;
4. Viola P., Jones M. Rapid object detection using a boosted cascade of simple features // Computer Vision and Pattern Recognition 2001
5. Viola P., Jones M. Fast Multi-view Face Detection // Proc. of Computer Vision and Pattern Recognition 2003
6. Viola P., Jones M. Robust Real-time Object Detection // International Journal of Computer Vision 2001
7. Freund Y., Schapire R.E. A decision-theoretic generalization of on-line learning and an application to boosting. In *Computational Learning Theory: Eurocolt '95*, pages 23–37. Springer-Verlag, 1995.
8. Smartphone OS Market Share (2015): http://www.idc.com/prodserv/smartphone-os-market-share.jsp
9. Swift, C++ Performance: http://www.primatelabs.com/blog/2014/12/swift-performance/
10. OpenCV modules: http://docs.opencv.org/3.1.0
11. Using C++ in Swift: https://developer.apple.com/library/ios/documentation/Swift/Conceptual/BuildingCocoaApps/index.html
12. CascadeClassifier Class Reference: http://docs.opencv.org/3.1.0/d1/de5/classcv_1_1CascadeClassifier.html
13. Viola-Jones Face Detection: https://sites.google.com/site/5kk73gpu2012/assignment/viola-jones-face-detection#TOC-Image-Pyramid

14. Song F., Tan X., Liu X., Chen S. Eyes Closeness Detection from Still Images with Multi-scale Histograms of Principal Oriented Gradients, Pattern Recognition, 2014.
15. Negative Samples. http://docs.opencv.org/2.4/doc/user_guide/ug_traincascade.html
16. Social Framework Reference: https://developer.apple.com/library/ios/documentation/Social/Reference/Social_Framework/
17. SwiftyVK: https://github.com/WE-St0r/SwiftyVK
18. Still and Video Media Capture: https://developer.apple.com/library/ios/documentation/AudioVideo/Conceptual/AVFoundationPG/Articles/04_MediaCapture.html
19. iPhone Hooks: https://www.instagram.com/developer/mobile-sharing/iphone-hooks/
20. Blink application repository: https://github.com/albertpod/blink